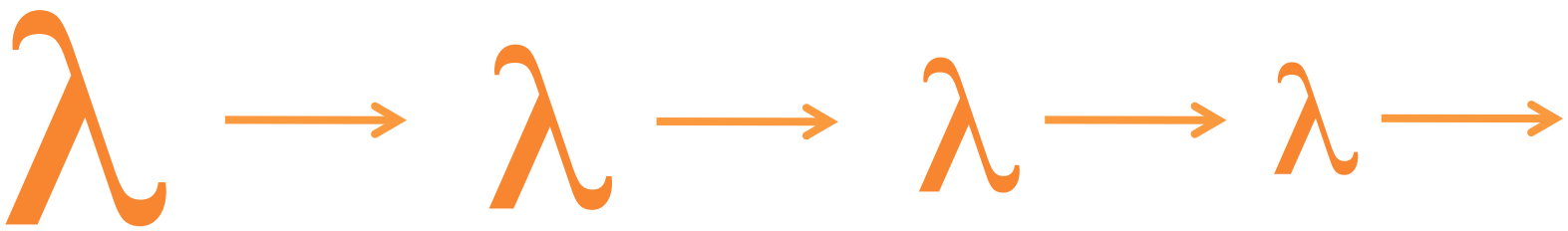
CS 4
Fundamentals of Computer Programming

Lecture 6: January 16, 2015

Higher-order functions, part 2



Last time

- Functions as arguments to other functions
- Abstracting around functional arguments
- Building more generic functions

Today

- Some new features of Scheme
- Functions as return values of functions
- Modifying the substitution model to deal with higher-order functions

Some new features of Scheme

`let` there be light

- Often want to define local values other than functions
- Could use internal **`define`**s
- Prefer to use a new special form: **`let`**

Usage

- Old way:

```
(define (foo x y)
  (define z (+ (* x x) (* y y))) ; not a function
  (sqrt (* (- z x) (- z y))))
```

- New way:

```
(define (foo x y)
  (let ((z (+ (* x x) (* y y))))
    (sqrt (* (- z x) (- z y)))))
```

Syntax and meaning

```
(let ((<var1> <expr1>)   ; <var>: variable
      (<var2> <expr2>)   ; <expr>: expression
      ...)

  <body>) ;; can use var1 and var2 here
```

- **`let`** is like a "multiple **`define`**"
  - can define multiple local names in a single **`let`**
  - convenient, but **`exprs`** must not depend on previous **`vars`**!

Syntax and meaning

```
(let ((<var1> <expr1>)   ; <var>: variable
      (<var2> <expr2>)   ; <expr>: expression
      ...)
  <body>) ;; can use var1 and var2 here
```

- Can have one or more **<var>**/**<expr>** pairs
- **<expr>** is evaluated, bound to **<var>**
- Substitute value of **<expr>** for **<var>** in **<body>**

Hmm, that sounds familiar…

- Substitute value of **`<expr>`** for **`<var>`** in **`<body>`**
- Where else have we seen this?
  - **`lambda`** expressions!

`let` is `lambda` in disguise!

```
(let ((var1 <expr1>)
      (var2 <expr2>))
  <body>)
```

- This is *syntactic sugar* for:

```
((lambda (var1 var2) <body>)
    <expr1> <expr2>)
```

- Substitution model is unchanged!

`let` is `lambda` in disguise!

Example:

```
(let ((x 1)
      (y 2))
  (+ x y))
```

- is syntactic sugar for:

```
((lambda (x y) (+ x y))
     1 2)
```

Why use `let`?

- Can evaluate an expression once and use multiple times
  - avoid unnecessary computations
- Can use it anywhere
  - don't need internal **`define`**s for local variables
  - no restrictions like with internal **`define`**s
- Can define multiple things in one **`let`**

`let` pitfall

- This doesn't work as you'd expect:

```
(let ((x (* 2 3))
      (y (* x 2)))
  (+ x y))
```

- Might expect this to return `18` (*i.e.* `6 + 12`)
- Actually, might result in an error, or a different value, depending on value of `x` *before* the `let` expression

`let` pitfall

```
(let ((x (* 2 3))
      (y (* x 2)))  ; not the x on prev line
  (+ x y))
```

- Rule: the values of the exprs are evaluated using values that existed *outside* the **`let`**
  - Desugaring **`let`** → **`lambda`** will show why this is the case

`let` pitfall

- If you want to achieve the same effect, write nested let expressions:

```
(let ((x (* 2 3)))
  (let ((y (* x 2)))
    (+ x y)))
```

`let` pitfall

- An alternative way is to use a related special form called **`let*`**:

```
(let* ((x (* 2 3))
       (y (* x 2)))
  (+ x y)) ; result: 18
```

- **`let*`** is equivalent to writing nested **`let`**s (previous slide)

display

- To print out something in Scheme, use **display** and **newline**:

```
(display "hello!") ; prints "hello!"
(newline)          ; advances to next line
```

- Other ways exist too (non-standard)
  - *e.g.* DrRacket has a **printf** function
- *Note:* **display** and **newline** are *not* special forms!

begin

- Might want to print something in an **if** statement:

```
(if (= x 10)
    ;; want to print x and return 2*x...
    ;; can't do this yet!
```

- Clauses of **if** only consist of *one* expression
  - may need to do more than one thing

begin

- Can use the **begin** special form to group many expressions into one:

```
(if (= x 10)
    (begin
        (display x)
        (newline)
        (* x 2))
    (/ x 2)) ; else clause
```

if x = 10, do all this

begin

- **begin** evaluates one or more expressions, returning the result of the last one only

```
(begin
   (display x)   ; print x
   (newline)     ; move to next line
   (* x 2))      ; return x * 2
```

- Right now, mainly useful for printing/ debugging
  - later will be useful for much more

`lambda` and implicit `begin`

- `lambda` expressions have an implicit `begin` in their body:

```
(define (print-and-square x)
  (display x)   ; implicit begin
  (newline)     ;
  (* x x))      ;
```

- Same as...

`lambda` and implicit `begin`

- `lambda` expressions have an implicit `begin` in their body:

```
(define (print-and-square x)
  (begin  ; not needed but OK
    (display x)
    (newline)
    (* x x)))
```

`lambda` and implicit `begin`

- Implicit **`begin`** in **`lambdas`** explains why internal procedures work correctly
- **`lambda`** expressions can contain any number of expressions in their bodies
  - including internal procedures

Main topic:

Returning functions from functions

Math:
Operators as return values

- You've probably seen:

$$f(x) = \frac{d}{dx}(F(x))$$

Math: Operators as return values

- The derivative operator
  - takes in…
    - a function
      - (from numbers to numbers)
  - returns…
    - *another* function
      - (from numbers to numbers)
    - representing how fast the first function changes at any point (*i.e.* how much *F(x)* changes as *x* changes)

$$f(x) = \frac{d}{dx}(F(x))$$

Math: Operators as return values

- You've probably also seen:

$$F(x) = \int f(x)dx$$

Math: Operators as return values

- The (indefinite) integration operator
  - takes in…
    - a function
      - from numbers to numbers
    - (and a value of the resulting function at some point
      e.g. F(0) = 0)
  - returns
    - a different function from numbers to numbers

$$F(x) = \int f(x)dx$$

Returning operators

- So *operators* (functions) can be return values, as well:

$$f(x) = \frac{d}{dx}(F(x))$$

$$F(x) = \int f(x)dx$$

Further motivation

- Besides mathematical operations that inherently return functions…
- …it's often nice, when designing programs, to have functions that create other functions with a particular structure

An example:

- Consider defining all these functions:

```
(define add1 (lambda (x) (+ x 1)))
(define add2 (lambda (x) (+ x 2)))
(define add3 (lambda (x) (+ x 3)))
(define add4 (lambda (x) (+ x 4)))
(define add5 (lambda (x) (+ x 5)))
```

- …repetitive, tedious.

The D.R.Y. principle

- D.R.Y. → "Don't Repeat Yourself"
- Whenever we find ourselves doing something rote/repetitive… ask:
- *Is there a way to abstract this?*
- Here, "abstract" means:
  - capture common features of old procedures in a *more general* new procedure

Abstracted adder function

- Generalize:

```
(define add1  (lambda (x) (+ x 1)))
(define add2  (lambda (x) (+ x 2)))
(define add3  (lambda (x) (+ x 3)))
...
```

- to:

```
(define (make-addn n) (lambda (x) (+ x n)))
```

Abstracted adder function

- Generalize to a function that can *create* adders:

```
(define (make-addn n)
  (lambda (x) (+ x n)))
```

- Equivalently (desugared):

```
(define make-addn
  (lambda (n)
    (lambda (x) (+ x n))))
```

- Note the nested lambda expressions!

How do I use it?

```
(define (make-addn n)
   (lambda (x) (+ x n)))
```

- `(define add2 (make-addn 2))`
- `(define add3 (make-addn 3))`
- `(add3 4)`
- `7`

Note:

- Now making new adders is a snap:
- Before:

```
(define add5 (lambda (x) (+ x 5)))
```

- Now:

```
(define add5 (make-addn 5))
```

- Less to think about / go wrong

Evaluating…

- `(define add3 (make-addn 3))`
  - Evaluate `(make-addn 3)`
    - evaluate `3` → `3`
    - evaluate `make-addn`

      → `(lambda (n) (lambda (x) (+ x n)))`
    - apply `make-addn` to `3`…
      - substitute `3` for `n` in `(lambda (x) (+ x n))`

      → `(lambda (x) (+ x 3))`
  - Make association:
    - `add3` bound to `(lambda (x) (+ x 3))`

Evaluating `(add3 4)`

- `(add3 4)`
- Evaluate `4` → `4`
- Evaluate `add3`

  → `(lambda (x) (+ x 3))`
- Apply `(lambda (x) (+ x 3))` to `4`
  - substitute `4` for `x` in `(+ x 3)`

  → `(+ 4 3)`

  → `7`

`make-addn`'s "signature"

```
(define (make-addn n)
  (lambda (x) (+ x n)))
```

- Takes in a numeric argument `n`
- Returns a function…
  - …which has, within it, a value "pre-substituted" for `n`
- Standard substitution model holds
  - with one small clarification...

Day 2

Evaluating a function call

To evaluate a function call…

1. Evaluate the operands (arguments)
2. Evaluate the operator (function)
3. Apply the function to its arguments

To apply a function call…

Clarify

1. Substitute the function argument *variables* with the *values* given in the call everywhere they occur
2. Evaluate the resulting expression

Clarify Substitution Model

- Substitute the function argument variables (*e.g.* `n`) with the values given in the call everywhere they occur
- *i.e.* "deep substitution"
- Happily plow through inner expressions, etc., **except**:
- Do **not** substitute for a variable inside any nested lambda expression that also uses the **same** variable as one of **its** arguments

Huh?

- "Do **not** substitute for a variable inside any nested lambda expression that also uses the **same** variable as one of **its** arguments."
- Idea: `lambda` "protects" its arguments from being substituted into
- I call this…
- ***"The Lambda Shield"***

The lambda shield

(expression involving x)

$\lambda(x)$

attempt to substitute value into x

- Cannot substitute for x in body of this lambda expression
  - since x is an argument of the lambda
- Other substitutions will succeed

Example

```
(define weird-protection-example
  (lambda (n)
    (lambda (n) (+ n n))))

(define foo (weird-protection-example 3))
```

- Should bind `foo` to `???`

Example

```
(define weird-protection-example
  (lambda (n)
    (lambda (n) (+ n n))))
(weird-protection-example 3)
```

→ Apply `(lambda (n) (lambda (n) (+ n n)))` to `3`

→ Substitute `3` for `n` in `(lambda (n) (+ n n))`

- Gives what?

Example

- → Apply `(lambda (n) (lambda (n) (+ n n)))` to `3`
- → Substitute `3` for `n` in `(lambda (n) (+ n n))`
- Gives what?
  - `(lambda (3) (+ 3 3))  ;; ??? nonsense!`
  - `(lambda (n) (+ 3 3))  ;; nope!`
  - `(lambda (n) (+ n n))  ;; correct!`
- The *lambda shield* protects `n` argument from being substituted into

NOTE!

- The lambda shield protects variables in a lambda expression from substitution when an *outer* lambda is being applied to its arguments
- When a lambda expr is applied to *its own* arguments, then substitution (obviously) happens
  - no shield
- Shielding only happens with nested lambdas

Examples

- Apply `(lambda (x) (+ x x))` to `3`
  - → substitute `3` for `x` in `(+ x x)` (no shielding)
  - → `(+ 3 3)` → `6`
- Apply `(lambda (x) (lambda (y) (+ x y)))` to `3`
  - → substitute `3` for `x` in `(lambda (y) (+ x y))`
  - → `(lambda (y) (+ 3 y))` (shielding not needed)
- Apply `(lambda (x) (lambda (x) (+ x x)))` to `3`
  - → substitute `3` for `x` in `(lambda (x) (+ x x))`
  - → `(lambda (x) (+ x x))` (`x` is shielded)

Why?

- Apply `(lambda (x) (lambda (x) (+ x x)))` to `3`
  - → substitute `3` for `x` in `(lambda (x) (+ x x))`
  - → `(lambda (x) (+ x x))` (`x` is shielded)
- *Why* do we have lambda shielding?
- A function like `(lambda (x) (+ x x))` should have a meaning which is independent of its context
  - should always mean "a function which doubles its input"
- Without shielding, this wouldn't be the case

Another Example

```
(define select-op
   (lambda (b)
      (if b
          (lambda (a b) (and a b))
          (lambda (a b) (or a b)))))
```

Example

```
(define select-op
 (lambda (b)
  (if b
   (lambda (a b)
     (and a b))
   (lambda (a b)
     (or a b)))))
```

- `(select-op #t)`

→ `(lambda (a b) (and a b))`

- **not**

  `(lambda (a b) (and a #t))`

- `(select-op #f)`

→ `(lambda (a b) (or a b))`

- **not**

  `(lambda (a b) (or a #f))`

Pop quiz (digression)

```
(define select-op
  (lambda (b)
    (if b
        (lambda (a b) (and a b))
        (lambda (a b) (or a b)))))
```

- Why not this?

```
(define select-op
  (lambda (b)
    (if b
        and
        or)))
```

Blast from the future...

- This way of using the argument to one function (`select-op`) to pick another function to be executed will be seen later in the course
- We'll use this idea to build a simple but powerful version of object-oriented programming inside Scheme

Note: nested `let` expressions

- `let` is just `lambda`, so...
- nested `let` expressions can shield variables just like nested lambda expressions

```
(let ((x 42))
   (let ((x 25)) ; new x
     (* x 2)))
```

→ `50`, not `84`

Note: nested `let` expressions

```
(let ((x 42))
  (let ((x 25))
    (* x 2)))
```

- Here we say that inner **x** "shadows" the outer **x**
- The outer **x** has no effect

Summation

- Consider again:

$$a(n) = \sum_{x=0}^{n} f(x)$$

To Scheme

```
(define (sum f low high)
  (if (> low high)
      0
      (+ (f low)
         (sum f (+ low 1) high))))
```

- Let's say **f**, **low** are usually fixed but **high** varies a lot
- Want to abstract around **high** only:

```
(define (make-sum f low)
  (lambda (high)
    (sum f low high)))
```

Idea

- We can use the ability to return lambda expressions to partially evaluate complex expressions to give simpler expressions
  - fill in the values that don't change
    - `low`, `(lambda (x) (* x x))`
  - leave a way to input values that change
    - `high`

To use…

```
(define (make-sum f low)
  (lambda (high)
    (sum f low high)))

(define sum-squares-to-n
  (make-sum (lambda (x) (* x x)) 0))
```

Result

```
(define (make-sum f low)
  (lambda (high)
    (sum f low high)))

(define sum-squares-to-n
  (make-sum (lambda (x) (* x x)) 0))
```

- **`sum-squares-to-n`** ends up bound to:

```
(lambda (high)
  (sum (lambda (x) (* x x)) 0 high))
```

  - where "`n`" is **`high`**

Calling `sum-squares-to-n`

- `(sum-squares-to-n 5)`

→ `55`

- `(sum-squares-to-n 10)`

→ `385`

- `(sum-squares-to-n 20)`

→ `2870`

... etc.

Taking it further...

```
(define multi-stage-add
  (lambda (a)
    (lambda (b)
      (lambda (c)
        (+ a b c)))))
```

- **`multi-stage-add`** is a function that takes a number and returns...

Taking it further...

```
(define multi-stage-add
  (lambda (a)
    (lambda (b)
      (lambda (c)
        (+ a b c))))))
```

- ... a function that takes a number and returns ...

Taking it further...

```
(define multi-stage-add
  (lambda (a)
    (lambda (b)
      (lambda (c)
        (+ a b c)))))
```

- ... a function that takes a number and returns ...

Taking it further...

```
(define multi-stage-add
  (lambda (a)
    (lambda (b)
      (lambda (c)
        (+ a b c)))))
```

- ... a number!

Using `multi-stage-add`

```
(define multi-stage-add
  (lambda (a)
    (lambda (b)
      (lambda (c)
        (+ a b c)))))

(define add-3-b-c
  (multi-stage-add 3))
```

`add-3-b-c` gets bound to:

```
(lambda (b)
  (lambda (c) (+ 3 b c)))
```

Using `add-3-b-c`

`add-3-b-c` gets bound to:

```
(lambda (b)
   (lambda (c) (+ 3 b c)))
```

```
(define add-3-4-c
  (add-3-b-c 4))
```

`add-3-4-c` gets bound to:

```
(lambda (c) (+ 3 4 c))
```

Using `add-3-4-c`

`add-3-4-c` gets bound to:

```
(lambda (c) (+ 3 4 c))
```

```
(add-3-4-c 5)
→ (+ 3 4 5)
→ 12
```

Doing it all at once

- Can also do this:
- `(((multi-stage-add 3) 4) 5)`

→ `12`

- Applying each argument results in a new function
- except the last one
  - yields final answer (number)

Curried functions

- Functions defined like this:

```
(define multi-stage-add
  (lambda (a)
    (lambda (b)
      (lambda (c)
        (+ a b c)))))
```

- Are called "curried" functions
  - after Haskell Curry, a logician
- Curried functions have one **`lambda`** per argument

Curried functions

- When we start using the Ocaml language, we will use curried functions as a matter of course
  - In Ocaml, all functions of multiple arguments are automatically curried
  - This makes it very easy to define partially-applied versions of functions

Curried functions

- If functions were automatically curried in Scheme, we could define **`make-addn`** as follows:

```
(define (make-addn n) (+ n))
```

- Or even just:

```
(define make-addn +)
```

What we've seen

- Functions as arguments

```
(define (sum f low high) (+ (f low) (sum ...
```

- Functions as return values

```
(define (make-addn n) (lambda (x) (+ x n)))
```

  - also derivative, integral…

- Functions as specialized versions of functions

```
(define (make-sum f low)
  (lambda (high) (sum f low high)))
```

- Functions which return functions which return functions...
  - **`multi-stage-add`**

How is this useful?

- Modeling math functions
  - *e.g.* $\frac{d}{dx}, \int f(x)dx$
- Partial evaluation of functions
  - often convenient
- Can build object-oriented programming system out of this
  - as we'll see

Constrast

- Functions which take other functions as arguments often used to make *more general* versions of a function
- Functions which return functions are often used to make *more specific* versions of a function
  - some values pre-substituted *i.e.* partially evaluated
- Because functions are data in Scheme, we get the power of both kinds of abstraction

Summary

- We can abstract operations around functions as well as numbers
- We can “compute” functions just as we can compute numbers and booleans
- Provides great power to
  - express
  - abstract
  - formulate high-level techniques

Next time

- Creating and using compound data structures
  - `cons` pairs
  - lists